University of Zagreb
Faculty of Electrical Engineering and Computing. Jaime Rodriguez Roncero

Machine Learning 1 2024/2025

http://www.fer.unizg.hr/predmet/maclea1

Third lab assignment: Support Vector Machine and Non-parametric Methods

*Version: 1.0

Deadline: **15 December 2024, 23:59**

Submission rules

By submitting the exercise, you confirm the following points:

1. You did not receive help from another when solving the exercise;
2. You attributed parts of the code that were taken from the Internet by referencing them in comments;
3. You did not use parts of the code from the Internet that are specific to the laboratory exercise;
4. You have not used UI-assistants for coding such as GitHub Copilot (including generative UI tools such as ChatGPT).

Violation of any of the above rules is considered a misdemeanor and results in academic sanctions.

Instructions

Third lab assignment consists of eight tasks. Follow the instructions in the text cells below. Solving the lab assignment boils down to **supplementing this notebook**: inserting one or more cells **below** the text of the task, writing the appropriate code, and executing the cells.

Make sure you fully understand the code you've written. When submitting the assignment, you must be able to modify and re-execute your code at the request of the teaching assistant. Furthermore, you need to understand the theoretical basis of what you are doing, within the framework of what we covered in the lecture. Below some tasks you can also find questions that serve as guidelines for a better understanding of the material (**do not write** the answers to the questions in the notebook). Therefore, do not limit yourself only to solving the tasks, but feel free to experiment. This is precisely the purpose of these assignments.

You should do the assignment **independently**. You can consult others on the principle way of solving it, but ultimately you have to do the assignment yourself. Otherwise, the assignment makes no sense.

In [6]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification
```

In [7]:
```
def plot_2d_svc_problem(X, y, svc=None):
    '''
    Plots a two-dimensional labeled dataset (X,y) and, if SVC object is given,
    the decision surfaces (with margin as well).
    '''
    assert X.shape[1] == 2, "Dataset is not two-dimensional"
    if svc!=None :
        # Create a mesh to plot in
        r = 0.03  # mesh resolution
        x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
        y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, r),
                             np.arange(y_min, y_max, r))
        XX=np.c_[xx.ravel(), yy.ravel()]
        Z = np.array([svc_predict(svc, x) for x in XX])
        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        plt.contourf(xx, yy, Z, cmap=plt.cm.Pastel1)

    # Plot the dataset
```

```
    plt.scatter(X[:,0],X[:,1], c=y, cmap=plt.cm.Paired, marker='o', s=50)

def svc_predict(svc, x) :
    h = svc.decision_function([x])
    if np.isclose(h, 0, atol=0.03):
        return 5
    elif (h >= -1 and h < -0.03) or (h > 0.03 and h <= 1):
        return 0.5
    else:
        return max(-1, min(1, h))

def plot_error_surface(err, c_range=(0,5), g_range=(0,5)):
    c1, c2 = c_range[0], c_range[1]
    g1, g2 = g_range[0], g_range[1]
    plt.xticks(range(0,g2-g1+1,5),range(g1,g2+1,5)); plt.xlabel("gamma")
    plt.yticks(range(0,c2-c1+1,5),range(c1,c2+1,5)); plt.ylabel("C")
    p = plt.contour(err);
    plt.imshow(1-err, interpolation='bilinear', origin='lower',cmap=plt.cm.gray)
    plt.clabel(p, inline=1, fontsize=10)

def plot_2d_clf_problem(X, y, h=None):
    '''
    Plots a two-dimensional labeled dataset (X,y) and, if function h(x) is given,
    the decision surfaces.
    '''
    assert X.shape[1] == 2, "Dataset is not two-dimensional"
    if h!=None :
        # Create a mesh to plot in
        r = 0.03  # mesh resolution
        x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
        y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
        xx, yy = np.meshgrid(np.arange(x_min, x_max, r),
                             np.arange(y_min, y_max, r))
        XX=np.c_[xx.ravel(), yy.ravel()]
        try:
            Z_test = h(XX)
            if Z_test.shape == ():
                # h returns a scalar when applied to a matrix; map explicitly
                Z = np.array(list(map(h,XX)))
            else :
                Z = Z_test
```

```
        except ValueError:
            # can't apply to a matrix; map explicitly
            Z = np.array(list(map(h,XX)))
        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        plt.contourf(xx, yy, Z, cmap=plt.cm.Pastel1)

    # Plot the dataset
    plt.scatter(X[:,0],X[:,1], c=y, cmap=plt.cm.tab20b, marker='o', s=50);

def knn_eval(n_instances=100, n_features=2, n_classes=2, n_informative=2,
             test_size=0.3, k_range=(1, 20), n_trials=40):

    train_errors = []
    test_errors = []
    ks = list(range(k_range[0], k_range[1] + 1))

    for i in range(0, n_trials):
        X, y = make_classification(n_instances, n_features, n_classes=n_classes,
                                   n_informative=n_informative, n_redundant=0, n_clusters_per_class=1)
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size)
        train = []
        test = []
        for k in ks:
            knn = KNeighborsClassifier(n_neighbors=k)
            knn.fit(X_train, y_train)
            train.append(1 - knn.score(X_train, y_train))
            test.append(1 - knn.score(X_test, y_test))
        train_errors.append(train)
        test_errors.append(test)

    train_errors = np.mean(np.array(train_errors), axis=0)
    test_errors = np.mean(np.array(test_errors), axis=0)
    best_k = ks[np.argmin(test_errors)]

    return ks, best_k, train_errors, test_errors
```

1. Support Vector Machine classifier (SVM)

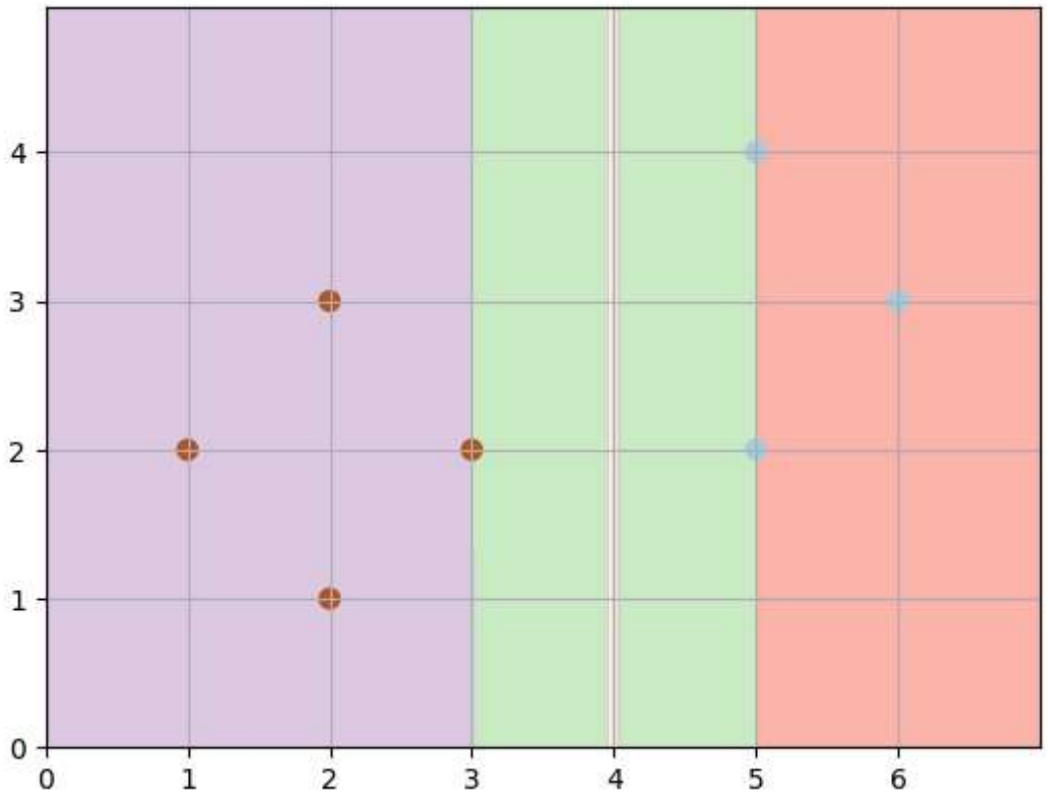
(a)

Familiarize yourself with the class `svm.SVC`, which implements an interface to the implementation of [`libsvm`](http://www.csie.ntu.edu.tw/~cjlin/libsvm/). Train the `SVC` model with a linear kernel function (i.e., without mapping of examples to feature space) on the data set `seven` (given below) with $N = 7$ examples. Print the coefficients w_0 and $\mathbf{w}$. Print the dual coefficients and support vectors. Finally, using the `plot_2d_svc_problem` function, plot the data, decision boundary and margin. The function accepts data, labels and a classifier (an object of class `SVC`) as arguments.

In [11]:
```
from sklearn.svm import SVC

seven_X = np.array([[2,1], [2,3], [1,2], [3,2], [5,2], [5,4], [6,3]])
seven_y = np.array([1, 1, 1, 1, -1, -1, -1])
```

In [12]:
```
model = SVC(kernel = 'linear')
model.fit(seven_X, seven_y)
print(f'w: {model.coef_}   b: {model.intercept_}')
print(f'Alpha: {model.dual_coef_}\nSupport vectors:\n {model.support_vectors_}\n')
plt.grid(True)
plot_2d_svc_problem(seven_X, seven_y, model)
```

```
w: [[-9.99707031e-01 -2.92968750e-04]]   b: [3.99951172]
Alpha: [[-4.99707031e-01 -1.46484375e-04  4.99853516e-01]]
Support vectors:
 [[5. 2.]
 [5. 4.]
 [3. 2.]]
```



Q: Which examples correspond to support vectors and why?

(b)

Define a `hinge(model, x, y)` function that calculates the hinge loss of the SVM model on `x` instance. Calculate the losses of the model trained on set `seven` for the examples $\mathbf{x}^{(2)} = (3, 2)$ and $\mathbf{x}^{(1)} = (3.5, 2)$ which are labeled as positive ($y = 1$) and for $\mathbf{x}^{(3)} = (4, 2)$ which is labeled as negative ($y = -1$). Also, calculate the average loss of the SVM on the set `seven`. Make sure the result is identical to what you would get using the built-in function `metrics.hinge_loss`.

In [16]:
```
from sklearn.metrics import hinge_loss

def hinge(model, x, y):
    decision_value = model.decision_function([x])[0]
```

```
    return max(0, 1 - y * decision_value)

print(f'Loss for x_1: {hinge(model, [3.5,2], 1)}',
    f'Loss for x_2: {hinge(model, [3,2], 1)}',
    f'Loss for x_3: {hinge(model, [4,2], -1)}',
    '-------------------------',
    sep = '\n')

N, D = seven_X.shape
print([hinge(model, seven_X[i], seven_y[i]) for i in range(N)])
average_hinge_loss_manual = np.average([hinge(model, seven_X[i], seven_y[i]) for i in range(N)])
print("Manual average hinge loss:", average_hinge_loss_manual)

average_hinge_loss_builtin = hinge_loss(seven_y, model.decision_function(seven_X))
print("Built-in hinge loss:", average_hinge_loss_builtin)
```

```
Loss for x_1: 0.5000488281249984
Loss for x_2: 0.00019531249999893419
Loss for x_3: 1.0000976562500012
-------------------------
[0, 0, 0, 0.00019531249999893419, 0.0003906250000014211, 0, 0]
Manual average hinge loss: 8.37053571429079e-05
Built-in hinge loss: 8.37053571429079e-05
```

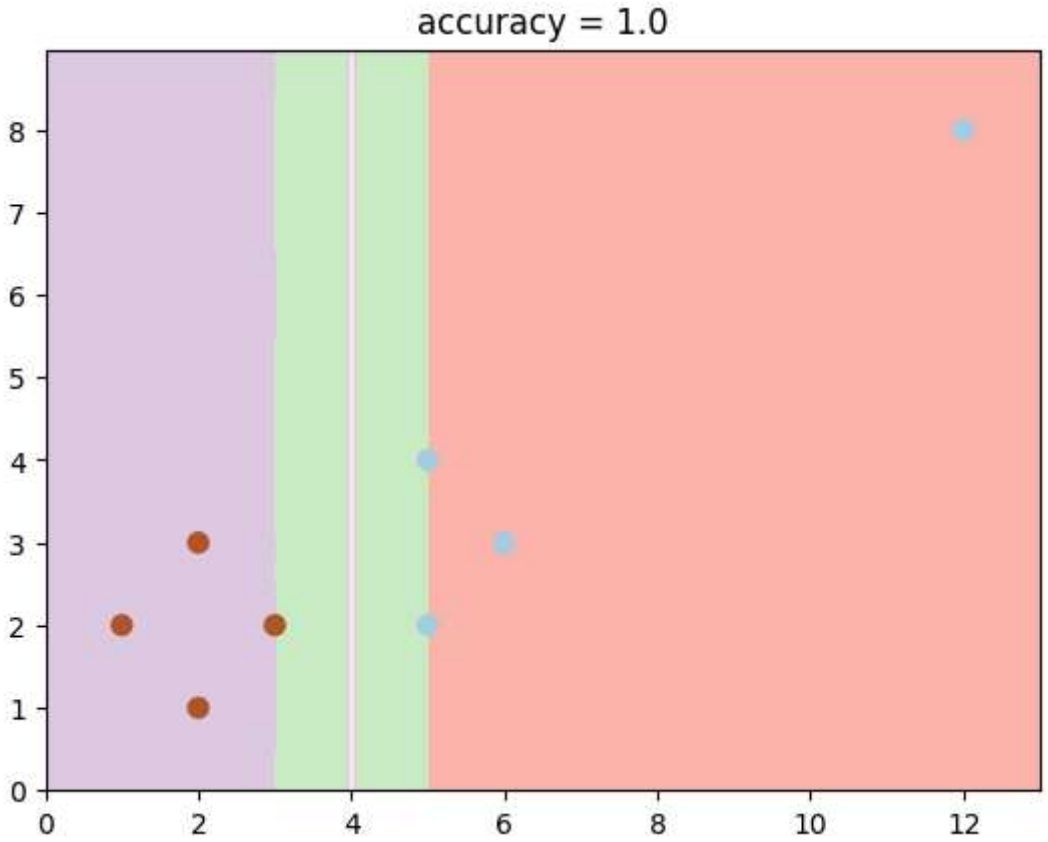
(c)

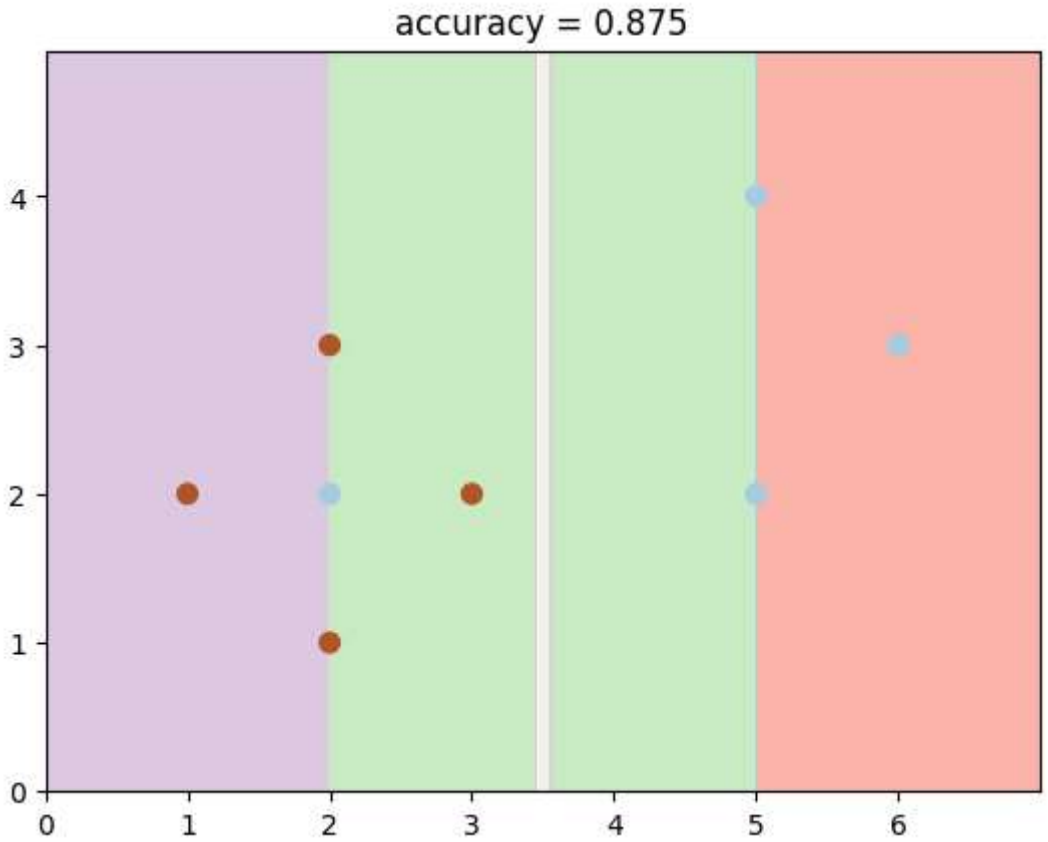
Now, we'll go back to the `outlier` ($N = 8$) and `unsep` ($N = 8$) datasets from the previous lab assignment (given below) and see how the SVM model handles them. Train a built-in SVM model (with a linear kernel) on this data and plot the decision boundary (together with the margin). Also print the accuracy of the model using the `metrics.accuracy_score` function.

In [19]:
```
from sklearn.metrics import accuracy_score

outlier_X = np.append(seven_X, [[12,8]], axis=0)
outlier_y = np.append(seven_y, -1)

unsep_X = np.append(seven_X, [[2,2]], axis=0)
unsep_y = np.append(seven_y, -1)
```

In [20]:
```
plt.figure()
model = SVC(kernel = 'linear')
model.fit(outlier_X, outlier_y)
plot_2d_svc_problem(outlier_X, outlier_y, model)
plt.title(f'accuracy = {accuracy_score(outlier_y, np.sign(model.decision_function(outlier_X)))}')

plt.figure()
model = SVC(kernel = 'linear')
model.fit(unsep_X, unsep_y)
plot_2d_svc_problem(unsep_X, unsep_y, model)
plt.title(f'accuracy = {accuracy_score(unsep_y, np.sign(model.decision_function(unsep_X)))}')
```

Out[20]:
```
Text(0.5, 1.0, 'accuracy = 0.875')
```



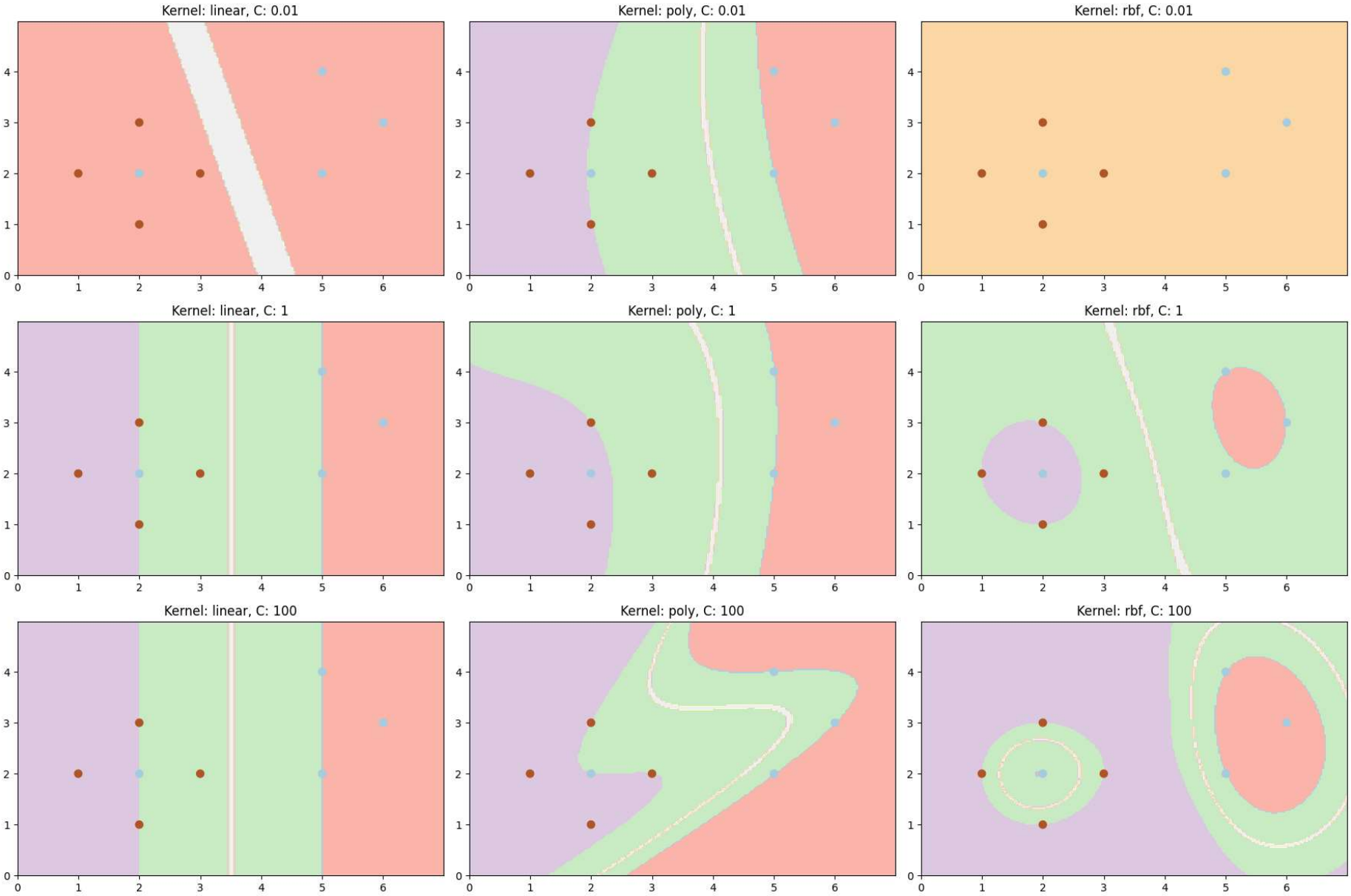


Q: How does an outlier affect SVM?
Q: How does a linear SVM handle a linearly inseparable data set?

2. Nonlinear SVM

This task will show how the choice of kernel affects the capacity of SVM. On the `unsep` set from the previous task, train three SVM models with different kernel functions: linear, polynomial, and radial basis function (RBF). Vary the hyperparameter C by the values $C \in \{10^{-2}, 1, 10^{2}\}$. For the other hyperparameters (the degree of the polynomial for the polynomial kernel and the hyperparameter γ for the RBF kernel) use the default values. Plot the boundaries between classes (and the margins) on a graph organized in a $3x3$ grid, where the columns represent different kernels and the rows represent different values of the C parameter.

In [24]:
```
fig, axes = plt.subplots(3, 3, figsize=(18, 12))

for i, C in enumerate([0.01, 1, 100]):
    for j, kernel in enumerate(["linear", "poly", "rbf"]):

        model = SVC(kernel=kernel, C=C)
        model.fit(unsep_X, unsep_y)
        plt.sca(axes[i,j])
        plot_2d_svc_problem(unsep_X, unsep_y, model)
        axes[i,j].set_title(f"Kernel: {kernel}, C: {C}")


plt.tight_layout()
plt.show()
```



3. Optimization of the SVM's hyperparameters

Together with the hyperparameter C, SVM model with the RBF kernel function has an additional hyperparameter $\gamma = \frac{1}{2\sigma^2}$ (precision). This parameter also determines the model's complexity: a large value for γ means that the RBF will be narrow, the examples will be mapped into a space where (according to the scalar product) they are very different from each other, which will result in more complex models. Conversely, a small value for γ means that the RBF will be wide, the examples will be more similar to each other, resulting in simpler models. This also means that, if we choose a larger γ, we need to regularize the model more strongly, i.e. we need to choose a smaller C, in order to prevent overfitting. For this reason, it is necessary to jointly optimize the hyperparameters C and γ, which is typically done by an exhaustive grid search. This approach is applied to all the models that contain more than one hyperparameter.

(a)

Define a function

> `grid_search(X_train, X_validate, y_train, y_validate, c_range=(c1,c2), g_range=(g1,g2), error_surface=False)`

which optimizes the parameters C and γ by grid search. The function should search over the hyperparameters $C \in \{2^{c_1}, 2^{c_1+1}, \ldots, 2^{c_2}\}$ and $\gamma \in \{2^{g_1}, 2^{g_1+1}, \ldots, 2^{g_2}\}$. The function should return the optimal hyperparameters (C^*, γ^*), i.e., those for which the model obtained the smallest error on the validation set. Additionally, if `surface=True`, the function should return matrices (of type `ndarray`) of model errors (expectation of 0-1 loss) on the training set and the validation set. Each matrix is of dimension $(c_2 - c_1 + 1) \times (g_2 - g_1 + 1)$ (rows correspond to different values of C and columns to different values of γ).

In [29]:
```
from sklearn.metrics import accuracy_score, zero_one_loss

def grid_search(X_train, X_validate, y_train, y_validate, c_range=(0,5), g_range=(0,5), error_surface=False):

    C_values = [2**c for c in range(c_range[0], c_range[1] + 1)]
    gamma_values = [2**g for g in range(g_range[0], g_range[1] + 1)]

    # Initialize error matrices
    train_errors = np.zeros((len(C_values), len(gamma_values)))
    val_errors = np.zeros((len(C_values), len(gamma_values)))

    # grid search
    best_C = None
    best_gamma = None
```

```
    min_val_error = float('inf')

    for i, C in enumerate(C_values):
        for j, gamma in enumerate(gamma_values):
            # Train the SVM with current hyperparameters
            model = SVC(C=C, gamma=gamma, kernel='rbf')
            model.fit(X_train, y_train)

            # Compute training and validation errors
            train_error = zero_one_loss(y_train, model.predict(X_train))
            val_error = zero_one_loss(y_validate, model.predict(X_validate))

            train_errors[i, j] = train_error
            val_errors[i, j] = val_error

            # Update the best parameters
            if val_error < min_val_error:
                min_val_error = val_error
                best_C = C
                best_gamma = gamma

    if error_surface:
        return best_C, best_gamma, train_errors, val_errors
    else:
        return best_C, best_gamma
```

(b)

Using the `datasets.make_classification` function, generate **two** datasets of $N = 200$ examples: one with $n = 2$ dimensions and the other with $n = 100$ dimensions. Let the examples come from two classes, with two groups corresponding to each class (`n_clusters_per_class=2`), so that the problem is a bit more complex, i.e. more non-linear. Keep all features informative. Divide the set of examples into a training set and a validation set in a 1:1 ratio.

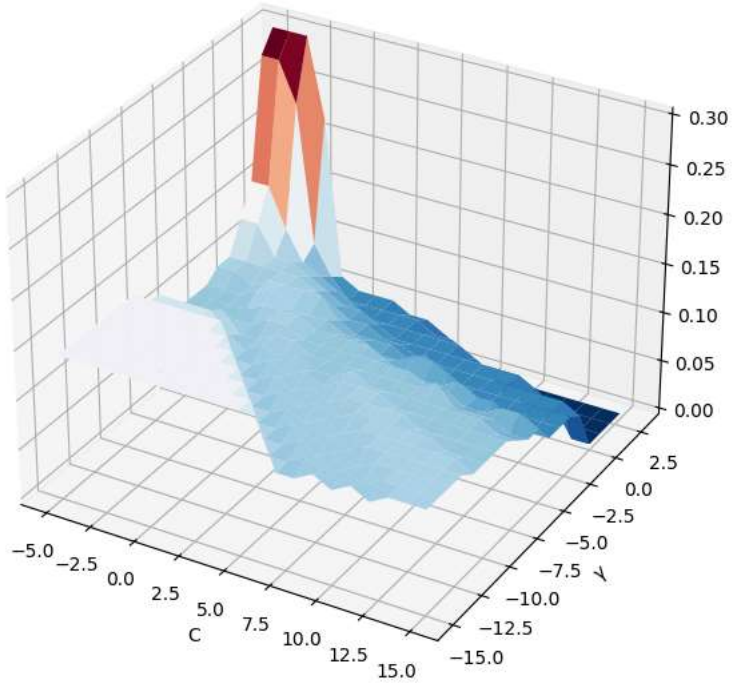
Using the both sets, optimize the SVM with the kernel function RBF, in the grid $C \in \{2^{-5}, 2^{-4}, \ldots, 2^{15}\}$ and $\gamma \in \{2^{-15}, 2^{-14}, \ldots, 2^{3}\}$. Plot the error surface of the model on the training set and the validation set, on both data sets (four plots in total) and print the optimal combinations of hyperparameters. You can use the `mlutils.plot_error_surface` function to display the error surface of the model.

In [32]:
```
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split


X_2D, y_2D = make_classification(
    n_samples=200,
    n_features=2,
    n_informative=2,
    n_redundant=0,
    n_clusters_per_class=2,
)


X_100D, y_100D = make_classification(
    n_samples=200,
    n_features=100,
    n_informative=100,
    n_redundant=0,
    n_clusters_per_class=2,
)

X_2D_train, X_2D_val, y_2D_train, y_2D_val = train_test_split(X_2D, y_2D, test_size=0.5)
X_100D_train, X_100D_val, y_100D_train, y_100D_val = train_test_split(X_100D, y_100D, test_size=0.5)
```

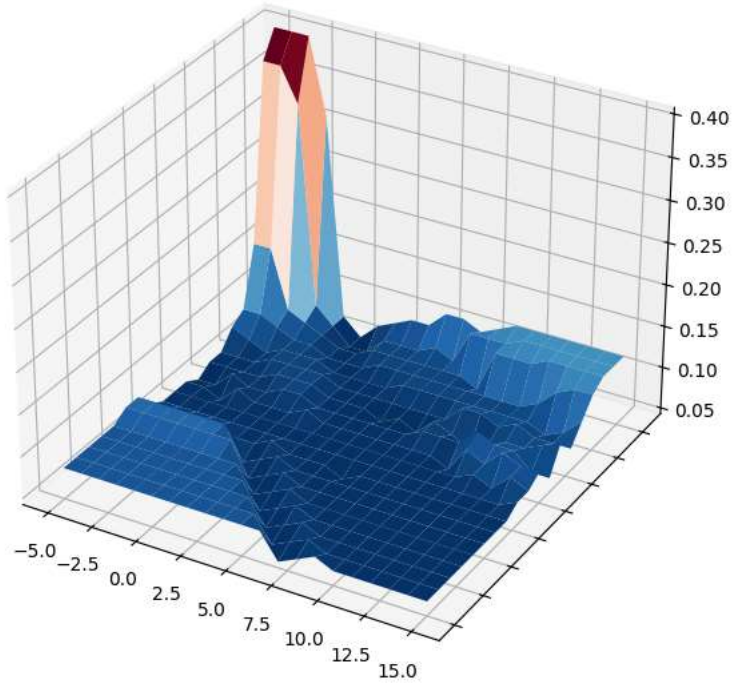
In [33]:
```
C = np.arange(-5, 16)  # log2(C)
g = np.arange(-15, 4)  # log2(Gamma)
gg, CC = np.meshgrid(g,C)

fig, ax = plt.subplots(2, 2, figsize=(18, 12), subplot_kw={'projection': '3d'}, sharey=True)

best_C, best_gamma, train_errors, val_errors = grid_search(X_2D_train, X_2D_val,
                                                           y_2D_train, y_2D_val,
                                                           c_range=(-5,15), g_range=(-15,3),
                                                           error_surface=True)

ax[0,0].plot_surface(CC, gg, train_errors, cmap='RdBu_r')
ax[0,0].set_title('Training errors for 2D dataset')
ax[0,0].set_xlabel("C")
ax[0,0].set_ylabel(r'$\gamma$')
```

```
ax[0,1].plot_surface(CC, gg, val_errors, cmap='RdBu_r')
ax[0,1].set_title('Validation errors for 2D dataset')

best_C, best_gamma, train_errors, val_errors = grid_search(X_100D_train, X_100D_val,
                                                           y_100D_train, y_100D_val,
                                                           c_range=(-5,15), g_range=(-15,3),
                                                           error_surface=True)

ax[1,0].plot_surface(CC, gg, train_errors, cmap='RdBu_r')
ax[1,0].set_title('training errors for 100D dataset')

ax[1,1].plot_surface(CC, gg, val_errors, cmap='RdBu_r')
ax[1,1].set_title('Validation errors for 100D dataset')

plt.tight_layout()
plt.show()
```

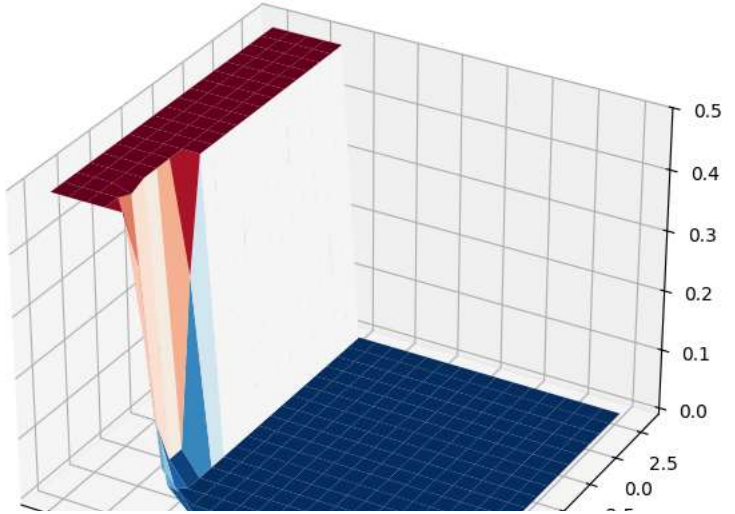
Training errors for 2D dataset



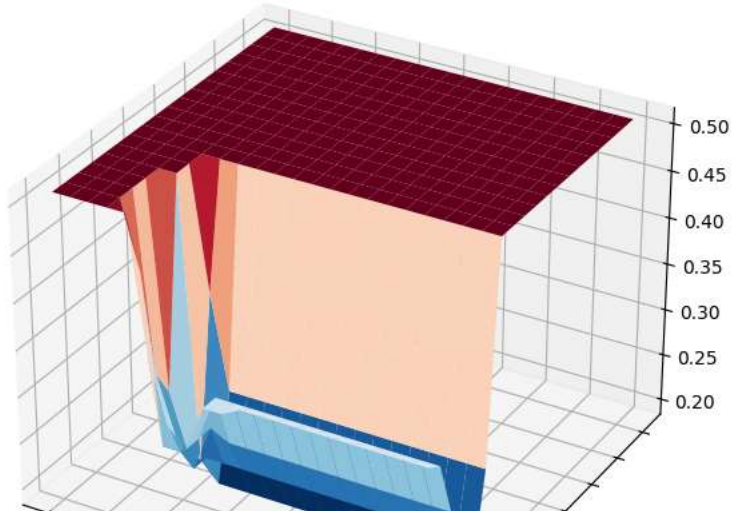
Validation errors for 2D dataset

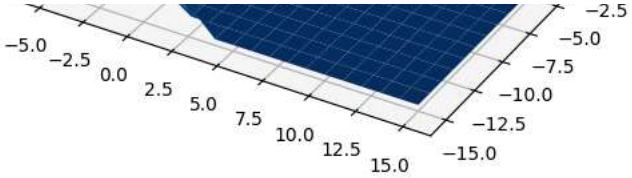


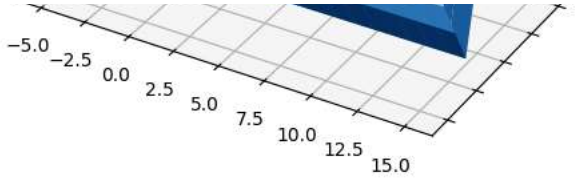
training errors for 100D dataset



Validation errors for 100D dataset







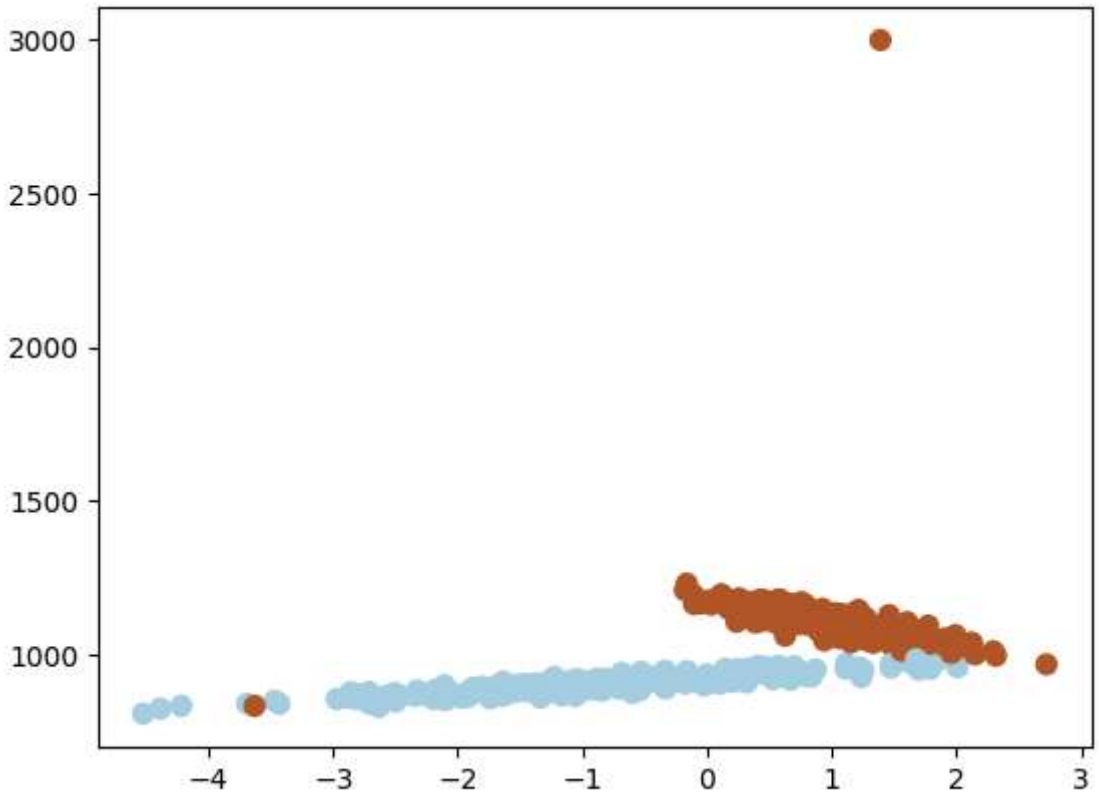
Q: Is the error surface different on the training set and the test set? Why?
Q: In the error surface plot, which part of the surface corresponds to overfitting and which part to underfitting? Why?
Q: How does the number of dimensions n affect the error surface, that is, the optimal hyperparameters (C^*, γ^*)?
Q: It is recommended that an increase in the value of γ should be accompanied by a decrease in the value of C. Do your results support that recommendation? Explain.

4. The effect of feature standardization in SVM

From the lab assignment on the topic of regression, we know that the features of different scales can make it impossible to interpret the learned linear regression model. However, this problem occurs with many models, so it is almost always important to scale the features before training, in order to prevent features with larger numerical ranges from dominating those with smaller numerical ranges. This also applies to SVM, where scaling can often significantly improve results. The goal of this task is to experimentally determine the influence of feature scaling on SVM accuracy.

We will now generate a two-class set of $N = 500$ examples with $n = 2$ features, so that the dimension x_1 has a larger values and a larger range than the dimension x_0, and we will add one example whose feature value x_1 jumps from other examples:

In [37]:
```
from sklearn.datasets import make_classification

X, y = make_classification(n_samples=500,n_features=2,n_classes=2,n_redundant=0,n_clusters_per_class=1, random_state=69)
X[:,1] = X[:,1]*100+1000
X[0,1] = 3000

plot_2d_svc_problem(X, y)
```

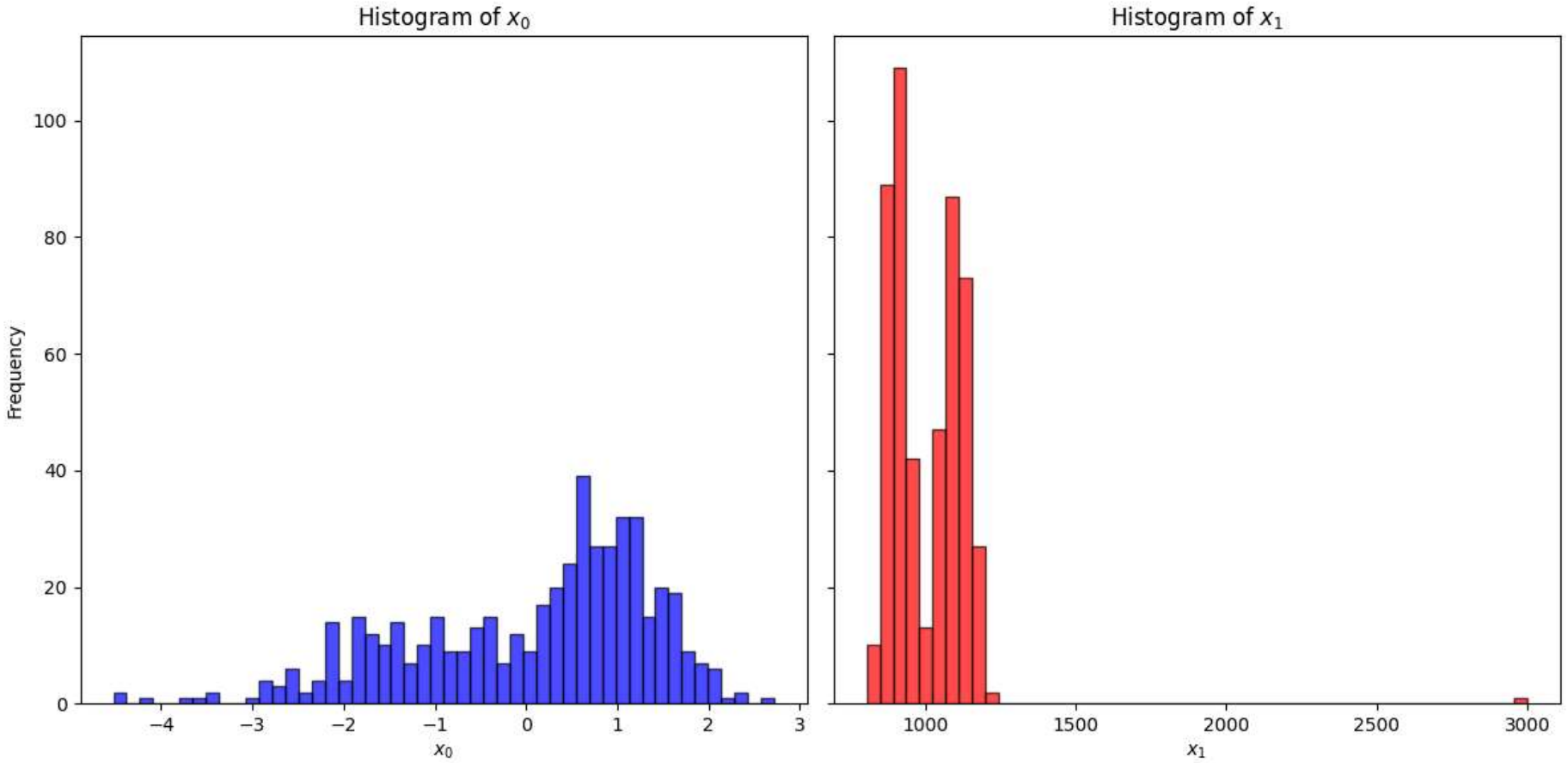


(a)

Familiarize yourself with the histogram plotting function `hist`. Display the histograms of the feature values x_0 and x_1 (use `bins=50` here and in the following tasks).

In [40]:
```
fig, axes = plt.subplots(1, 2, figsize=(12, 6), sharey=True)

axes[0].hist(X[:, 0], bins=50, color='blue', alpha=0.7, edgecolor='black')
axes[0].set_title('Histogram of $x_0$')
axes[0].set_xlabel('$x_0$')
axes[0].set_ylabel('Frequency')

axes[1].hist(X[:, 1], bins=50, color='red', alpha=0.7, edgecolor='black')
axes[1].set_title('Histogram of $x_1$')
```

```
axes[1].set_xlabel('$x_1$')

plt.tight_layout()
plt.show()
```

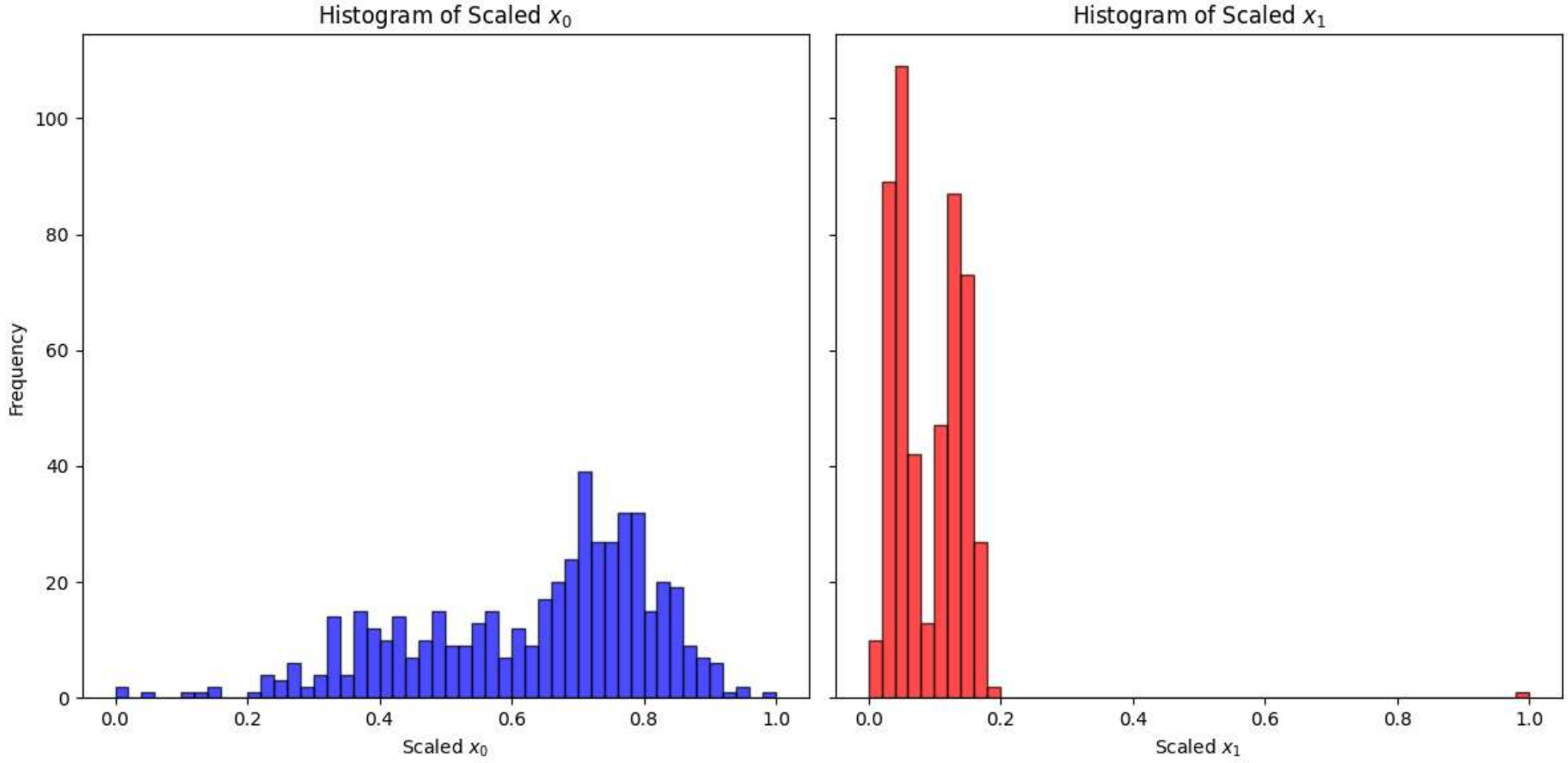


(b)

Take a look at the `preprocessing.MinMaxScaler` class. Display histograms of feature values x_0 and x_1 if they are scaled by min-max scaling (two histograms in total).

In [43]:
```
from sklearn.preprocessing import MinMaxScaler
```

```
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

fig, axes = plt.subplots(1, 2, figsize=(12, 6), sharey=True)

axes[0].hist(X_scaled[:, 0], bins=50, color='blue', alpha=0.7, edgecolor='black')
axes[0].set_title('Histogram of Scaled $x_0$')
axes[0].set_xlabel('Scaled $x_0$')
axes[0].set_ylabel('Frequency')

axes[1].hist(X_scaled[:, 1], bins=50, color='red', alpha=0.7, edgecolor='black')
axes[1].set_title('Histogram of Scaled $x_1$')
axes[1].set_xlabel('Scaled $x_1$')

plt.tight_layout()
plt.show()
```



Q: How does this scaling work?

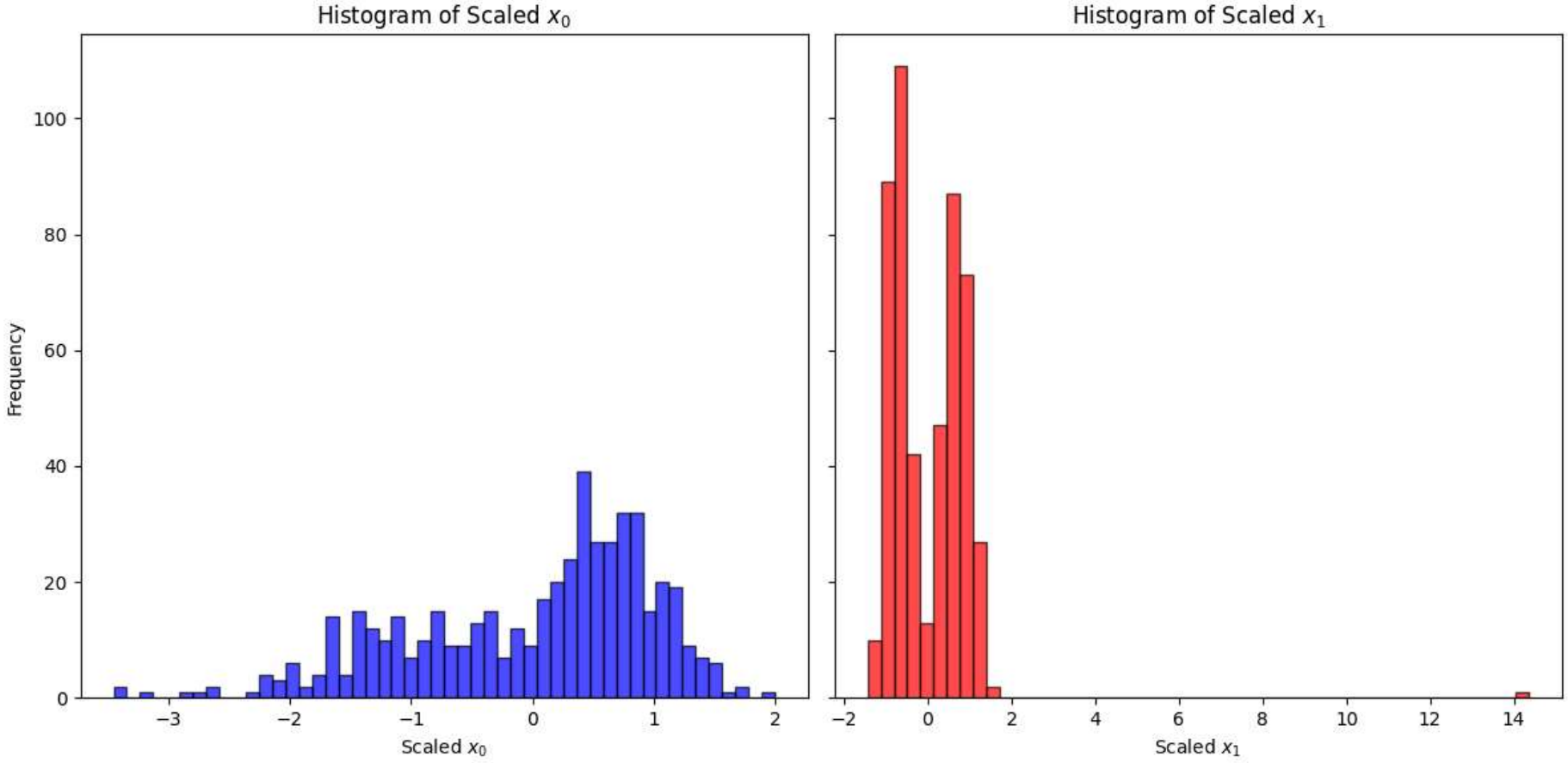
Q: The resulting histograms are very similar. What's the difference?

(c)

Take a look at the `preprocessing.StandardScaler` class. Display the histograms of the feature values x_0 and x_1 if they are scaled by standard scaling (two histograms in total).

In [47]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
```

```
X_scaled = scaler.fit_transform(X)

fig, axes = plt.subplots(1, 2, figsize=(12, 6), sharey=True)

axes[0].hist(X_scaled[:, 0], bins=50, color='blue', alpha=0.7, edgecolor='black')
axes[0].set_title('Histogram of Scaled $x_0$')
axes[0].set_xlabel('Scaled $x_0$')
axes[0].set_ylabel('Frequency')

axes[1].hist(X_scaled[:, 1], bins=50, color='red', alpha=0.7, edgecolor='black')
axes[1].set_title('Histogram of Scaled $x_1$')
axes[1].set_xlabel('Scaled $x_1$')

plt.tight_layout()
plt.show()
```



Q: How does this scaling work?

Q: The resulting histograms are very similar. What's the difference?

(d)

Divide the set of examples into a training set and a test set in a 1:1 ratio. Train the SVM with the RBF kernel function on the training set and calculate the accuracy of the model on the test set, using three variants of the above set: unscaled features, standardized features, and min-max scaling. Use default values for C and γ. Calculate the accuracy of each of the three models on the training set and the test set. Repeat the procedure several times (e.g. 30) and average the results (in each repetition generate the data as given at the beginning of this task).

NB: On the training set, the scaling parameters should first be calculated and then scaling should be applied (function `fit_transform`), while on the test set only scaling should be applied with the parameters obtained on the learning set (function `transform`).

In [51]:
```
reps = 30
samples = 500
results = {'unscaled': [], 'standard_scaled': [], 'minmax_scaled': []}

for _ in range(reps):
    X, y = make_classification(
        n_samples=samples, n_features=2, n_classes=2,
        n_redundant=0, n_clusters_per_class=1)
    X[:, 1] = X[:, 1] * 100 + 1000
    X[0, 1] = 3000
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5)

    # Unscaled
    model = SVC(kernel='rbf')
    model.fit(X_train, y_train)
    train_acc_unscaled = accuracy_score(y_train, model.predict(X_train))
    test_acc_unscaled = accuracy_score(y_test, model.predict(X_test))
    results['unscaled'].append((train_acc_unscaled, test_acc_unscaled))

    # Standard scaling
    standard_scaler = StandardScaler()
    X_train_standard = standard_scaler.fit_transform(X_train)
    X_test_standard = standard_scaler.transform(X_test)
    model = SVC(kernel='rbf')
    model.fit(X_train_standard, y_train)
    train_acc_standard = accuracy_score(y_train, model.predict(X_train_standard))
    test_acc_standard = accuracy_score(y_test, model.predict(X_test_standard))
    results['standard_scaled'].append((train_acc_standard, test_acc_standard))

    # Min-max scaling
    minmax_scaler = MinMaxScaler()
    X_train_minmax = minmax_scaler.fit_transform(X_train)
    X_test_minmax = minmax_scaler.transform(X_test)
    model = SVC(kernel='rbf')
    model.fit(X_train_minmax, y_train)
    train_acc_minmax = accuracy_score(y_train, model.predict(X_train_minmax))
```

```
    test_acc_minmax = accuracy_score(y_test, model.predict(X_test_minmax))
    results['minmax_scaled'].append((train_acc_minmax, test_acc_minmax))
```

In [52]:
```
averaged_results = {
    key: (np.mean([r[0] for r in val]), np.mean([r[1] for r in val]))
    for key, val in results.items()
}

for key, val in averaged_results.items():
    print(key, val)
```

```
unscaled (0.7709333333333331, 0.7646666666666667)
standard_scaled (0.9543999999999998, 0.9520000000000001)
minmax_scaled (0.9456, 0.9439999999999998)
```

Q: Are the results as expected? Explain.
Q: Would it be ok if we applied the `fit_transform` function to the entire data set? Why? Would it be ok if we applied the function separately to the training set and separately to the test set? Why?

5. k-nearest neighbors algorithm

In this task, we will analyse a simple classification model called **k-nearest neighbor algorithm**. First, you will implement it independently to familiarize yourself with the inner workings of this model, and then you will move on to the analysis of its hyperparameters (using a built-in class, for efficiency).

(a)

Implement the `KNN` class, which implements the k-nearest neighbors algorithm. An optional parameter of the constructor is the number of neighbors `n_neighbors` (k), with the default value set to 3. Define the methods `fit(X, y)` and `predict(X)`, which are used for training the model and prediction, respectively. As a distance measure, use Euclidean distance (`numpy.linalg.norm`; watch out for the `axis` parameter). It is not necessary to implement any weighting function.

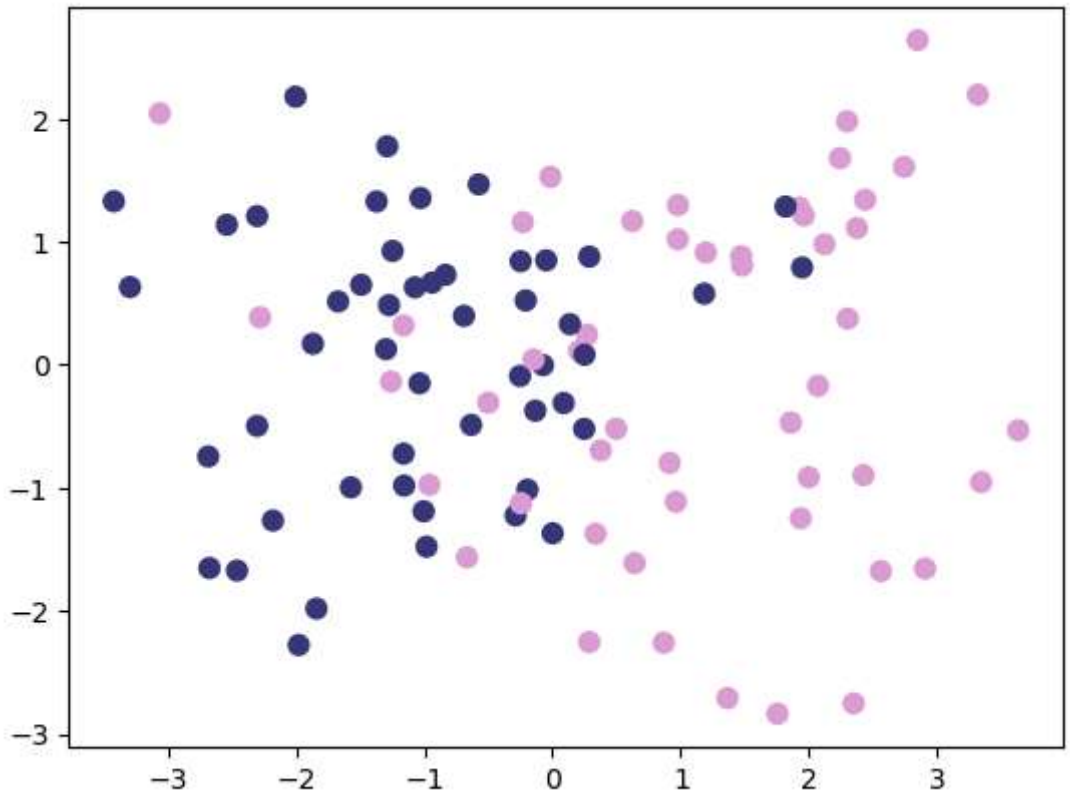
In [58]:
```
from numpy.linalg import norm

class KNN:
    def __init__(self, n_neighbors=3):
```

```
        self.n_neighbors = n_neighbors

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train

    def predict(self, X_test):
        predictions = []
        for x in X_test:
            norms = norm(self.X_train - x, axis=1)
            nearest_indices = np.argsort(norms)[:self.n_neighbors]
            nearest_labels = self.y_train[nearest_indices]
            prediction = np.bincount(nearest_labels).argmax()
            predictions.append(prediction)

        return np.array(predictions)
```

(b)

To make sure your implementation is correct, compare it with the `neighbors.KNeighborsClassifier` class. Since the mentioned class uses various optimization tricks when finding the nearest neighbors, make sure to set the `algorithm=brute` parameter, otherwise your predictions may differ. Compare models on a given (artificial) dataset (recall how arrays are compared; `numpy.all`).

In [61]:
```
from sklearn.datasets import make_classification
X_art, y_art = make_classification(n_samples=100, n_features=2, n_classes=2,
                                   n_redundant=0, n_clusters_per_class=2,
                                   random_state=69)
plot_2d_clf_problem(X_art, y_art)
```



In [62]:
```
from sklearn.neighbors import KNeighborsClassifier
X_train, X_test, y_train, y_test = train_test_split(X_art, y_art, test_size=0.5)
knn_implemented = KNN()
knn_implemented.fit(X_train, y_train)
y_pred_implemented = knn_implemented.predict(X_test)

knn_sklearn = KNeighborsClassifier(n_neighbors=3, algorithm='brute')
knn_sklearn.fit(X_train, y_train)
y_pred_sklearn = knn_sklearn.predict(X_test)

are_equal = np.all(y_pred_implemented == y_pred_sklearn)
print(are_equal)
```

```
True
```

6. The effect of the hyperparameter *k*

The k-nn algorithm has a hyperparameter k (number of neighbors). This hyperparameter directly affects the complexity of the algorithm, so it is extremely important to choose its value correctly. As with many other algorithms, with the k-nn algorithm the optimal value of the hypermeter k depends on the specific problem, including the number of examples N, the number of features (dimensions) n and the number of classes K.

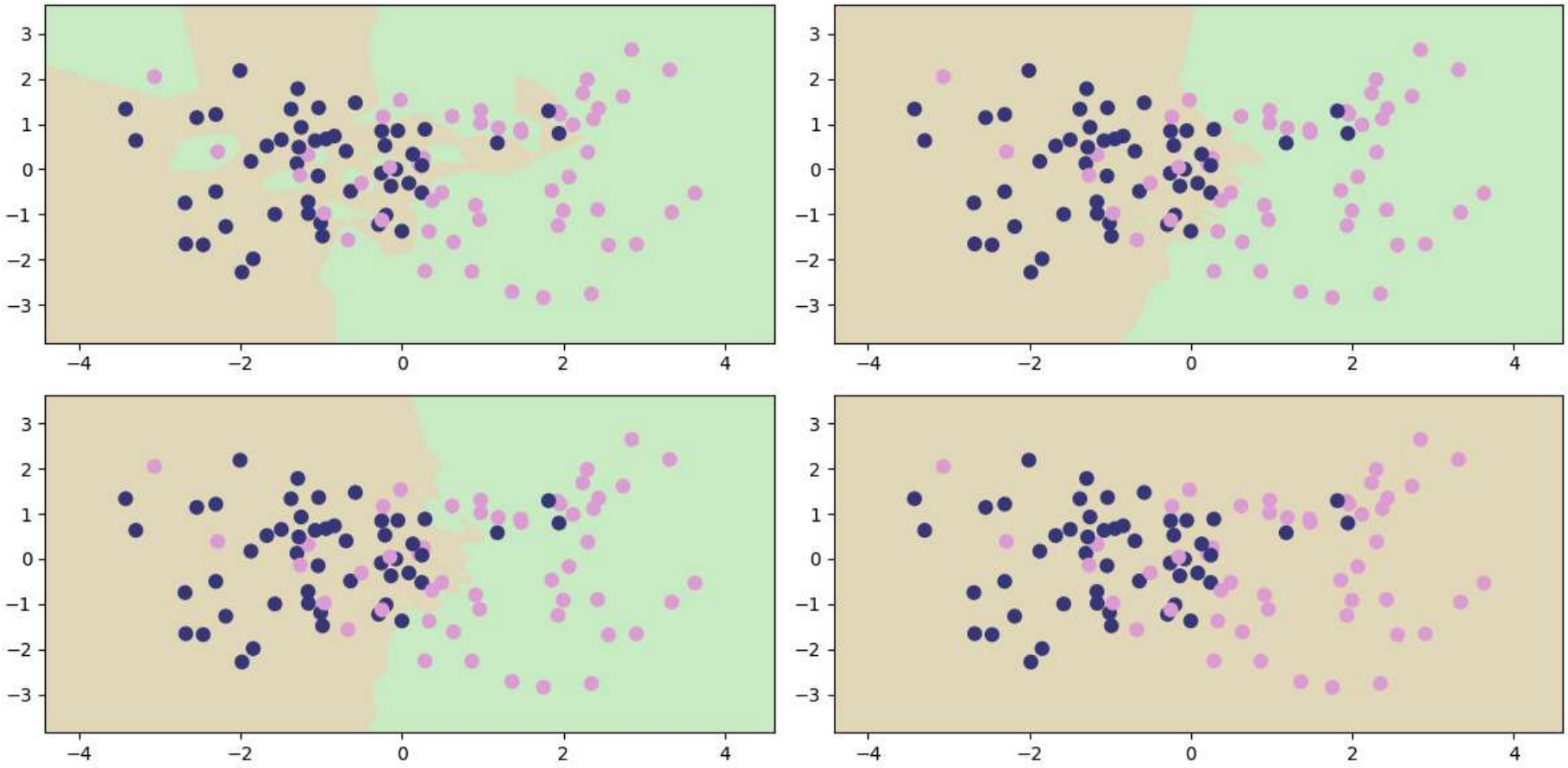
In order to obtain more reliable results, it is necessary to repeat some of the experiments on different data sets and then average the obtained error values. Use the function: `knn_eval` which trains and tests the k-nearest neighbors model on a total of `n_instances` of examples. For each value of the hyperparameter from the given interval `k_range` the function repeats `n_trials` measurements, generating for each measurement a new data set and dividing it into a training and a testing set. The test set ratio is defined by the `test_size` parameter. The return value of the function is a quadruple `(ks, best_k, train_errors, test_errors)`. The `best_k` value is the optimal value of the hyperparameter k (the value for which the error on the test set is the smallest). The `train_errors` and `test_errors` values are lists of errors on the training set and the testing set for all considered values of the hyperparameter k, while `ks` stores all the considered values of the hyperparameter k.

(a)

Using the data from the task 5, use the function `plot_2d_clf_problem` to plot the example space and the areas corresponding to the first and second class. Repeat this for $k \in [1, 5, 20, 100]$.

NB: The implementation of the `KNeighborsClassifier` algorithm from the `scikit-learn` package will probably work faster than your implementation, so use it in the remaining experiments.

In [66]:
```
fig, ax = plt.subplots(2, 2, figsize = (12, 6))

for i, k in enumerate([1, 5, 20, 100]):
    model = KNeighborsClassifier(n_neighbors=k)
    model.fit(X_art, y_art)
    plt.sca(ax[i//2, i%2])
    plot_2d_clf_problem(X_art, y_art, h=lambda x :model.predict(x) == 0)

plt.tight_layout()
```

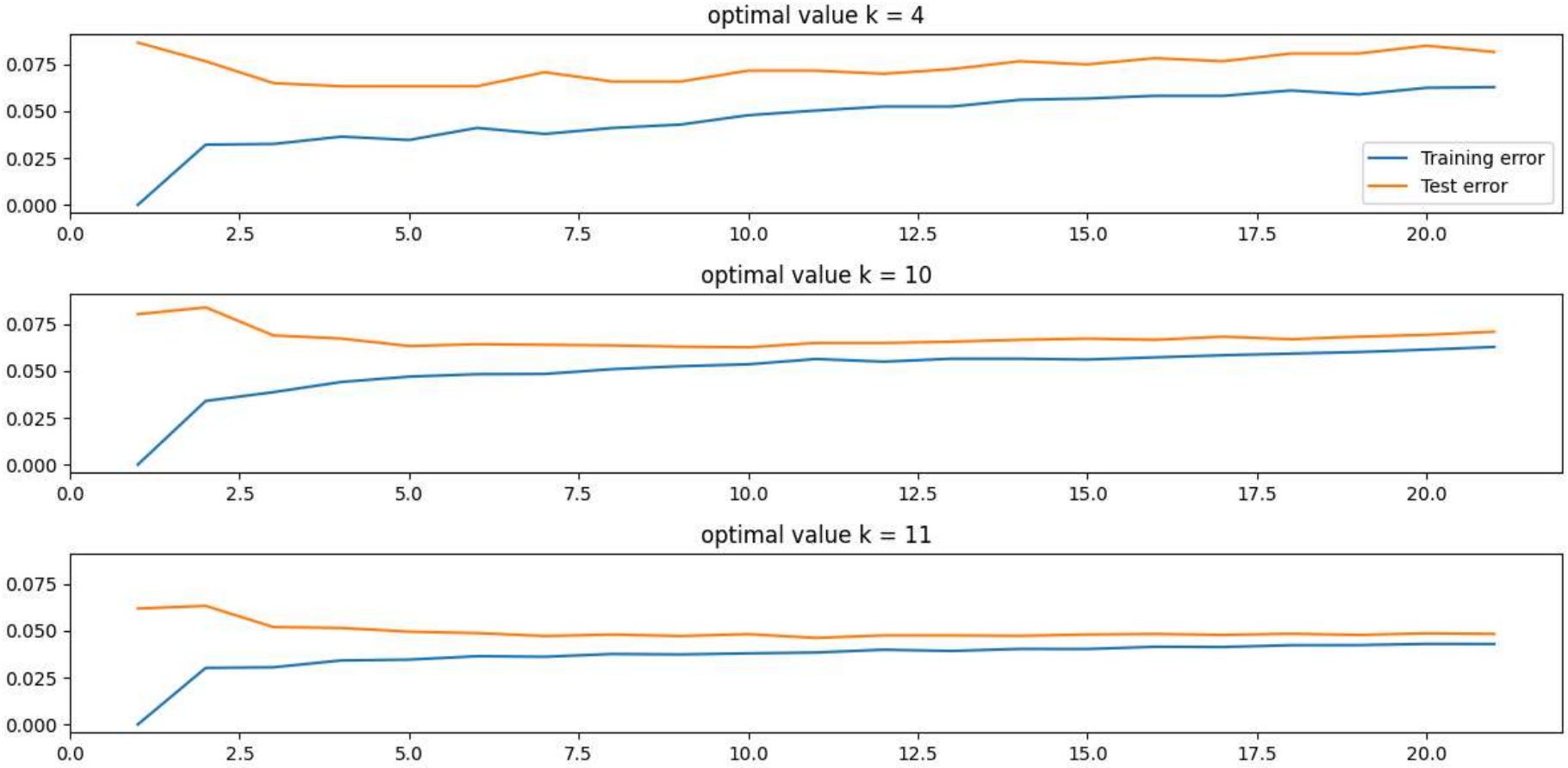


Q: How does k affect the border between the classes?
Q: How does the algorithm behave in extreme situations: $k = 1$ and $k = 100$?

(b)

Using the `knn_eval` function, plot the training and testing errors as a function of the hyperparameter $k \in \{1, \ldots, 20\}$, for $N = \{100, 250, 750\}$ examples. Make 3 separate graphs. For each graph, print the optimal value of the hyperparameter k (easiest as a plot title; see `plt.title`).

In [86]:
```
fig, ax = plt.subplots(3, 1, figsize = (12,6), sharey=True)

for i, N in enumerate([100, 250, 750]):
    ks, best_k, train_errors, test_errors = knn_eval(n_instances=N, k_range=(1, 21))
    ax[i].plot(ks, train_errors, label='Training error')
    ax[i].plot(ks, test_errors, label='Test error')
    ax[i].set_title(f'optimal value k = {best_k}')
    if i == 0:
        ax[i].legend()

plt.tight_layout()
```



Q: How does the optimal value of the hyperparameter k change with respect to the number of examples N? Why?
Q: Which area of the graph corresponds to overfitting and which to underfitting? Why?
Q: Is it always possible to reach an error of 0 on the training set?

7. Irrelevant features

We'd like to check to what extent the k-nearest neighbors algorithm is sensitive to the presence of irrelevant features. In order to do so, we can use the function [`datasets.make_classification`](http://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification .html) to generate a set of examples in which some of the features are irrelevant. Namely, the parameter `n_informative` determines the number of essential features, while the parameter `n_features` determines the total number of features. If `n_features > n_informative`, then some of the features will be irrelevant. Instead of using the `make_classification` function directly, we will use the `knn_eval` function, which only takes these parameters but allows us to make more reliable estimates.

Use the `mlutils.knn_eval` function in two ways. In both ways, use $N = 1000$ examples, $n = 10$ features, and $K = 5$ classes, but for the first let all 10 features be informative, and for the second let only 5 out of 10 features be informative. Print the training and testing errors for both models for the optimal value of k (the value for which the testing error is the smallest).

In [89]:
```
ks, best_k, train_errors, test_errors = knn_eval(n_instances=1000, n_features=10, n_classes=5, n_informative=10)
print(f'train error: {train_errors[best_k]}', f'test error: {test_errors[best_k]}', sep='\n')

ks, best_k, train_errors, test_errors = knn_eval(n_instances=1000, n_features=10, n_classes=5, n_informative=5)
print(f'train error: {train_errors[best_k]}', f'test error: {test_errors[best_k]}', sep='\n')
```

```
train error: 0.0905357142857143
test error: 0.12816666666666668
train error: 0.1710714285714286
test error: 0.20716666666666672
```

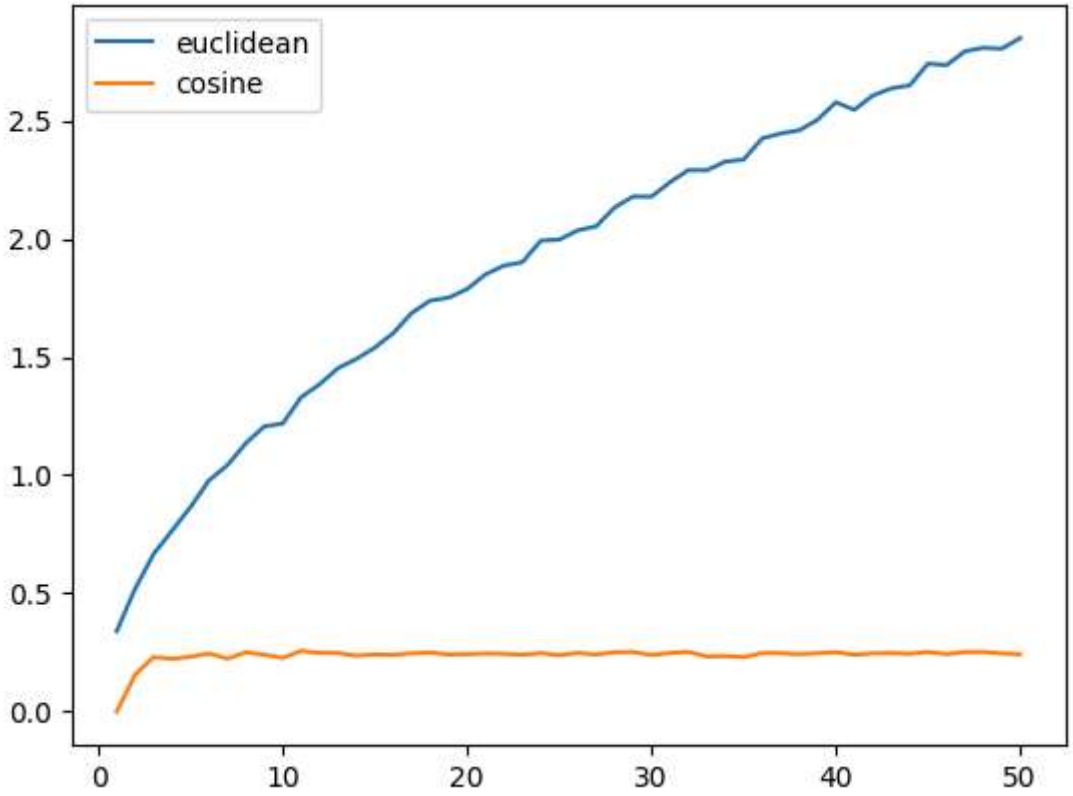
Q: Is the k-nearest neighbor algorithm sensitive to irrelevant features? Why?
Q: Is this problem present in other models that we have worked with so far (e.g., logistic regression)?
Q: How would a k-nearest neighbor model perform on a dataset with features of different scales? Please explain in detail.

8. "Curse of dimensionality"

The "curse of dimensionality" refers to a number of phenomena associated with high-dimensional spaces. In most cases, these phenomena, which are mostly counterintuitive, lead to the decrease of the model's accuracy as the number of dimensions (features) grows.

In general, the increase in the number of dimensions makes all the points in the input space become (in terms of Euclidean distance) increasingly distant from each other and, consequently, the differences in distances between the points are lost. We will experimentally verify that this is indeed the case. Study the function `metrics.pairwise_distances`. Generate 100 random vectors in different dimensions $n \in [1, 2, \ldots, 50]$ and calculate the *average* Euclidean distance between all the pairs of these vectors. To generate random vectors, use the function `numpy.random.random`. On the same graph, plot the curve for average cosine distances (change the value of the `metric` parameter).

In [92]:
```
from sklearn.metrics.pairwise import pairwise_distances

range = np.arange(1,51)
euclidean_average = np.zeros(len(range))
cosine_average = np.zeros(len(range))

for i, n in enumerate(range):
    random_vectors = np.random.random((100, n))
    euclidean_distances = pairwise_distances(random_vectors, metric='euclidean')
    cosine_distances = pairwise_distances(random_vectors, metric='cosine')
    euclidean_average[i] = np.average(euclidean_distances)
    cosine_average[i] = np.average(cosine_distances)

plt.plot(range, euclidean_average, label='euclidean')
plt.plot(range, cosine_average, label='cosine')
plt.legend()
plt.show()
```



Q: Try to explain the differences in results. Which of these two measures would you use to classify high-dimensional data?

Q: Why is this problem particularly pronounced with the k-nearest neighbor algorithm?